# BUILDING DATABASE SYSTEMS

*In this final installment of our series, Wilf Hey reviews the basic process of database design, and discusses functions within the* SR-Info *database software.*



## GLOSSARY

**DATABASE SYSTEM** – The software that provides the tools for creating, accessing, and maintaining database files; often the database system includes a program interpreter, so that it (rather than the operating system) is actually responsible for running the database programs.

**DATABASE FILE** – A collection of individual records, all of the same general format. In *SR-Info* a database file has the extension .DBF.

**RELATIONAL DATABASE** – A database system that makes it easy (or automatic) to use keys for linking related records from two or more database files.

**FIELD** – A unit of data assigned a name, and defined in terms of its size and type. Fields are grouped together to form the structure of a database file.

**SIZE** – The length of a field – the same (for that field) in every record in the database file, even when the field is empty.

**STRUCTURE** – The definition of what information is held within a database file; this is made up of fields. Each record in a particular database file has the same structure.

**KEY** – A field (or group of fields) that occurs in each record of a database file, used to make records easy to locate, or easy to group together. A key can also be used to put the records of a database file into a specific order for printing.

**INDEX** – A tiny file related to a key, used by the database system to pick records out of a database file. In *SR-Info*, index files have the extension name .NDX.

Database applications are the most common use of computers. A database system can be used for many jobs – from a government's statistical records stored on mainframe computers, to collecting favourite recipes on a home PC.

This series has given you the opportunity and the guidance necessary to build your own database, learning and applying the skills common to all database applications.

The dominant basic idea behind all database applications is the need for storing data in an organised way, so that information can be added to, adjusted, and consulted with maximum ease.

You need to develop two skills to make a working database; first, the design of database structures; second, writing the programs that enable you to use and maintain the database.

This series has been built in two parts, discussing these skills separately. In *Designing a Database* we have been looking at the design principles that apply to all database systems – whatever database package or language is used. In *Developing with SR-Info* we have looked at how those principles work in practice, using the *SR-Info* database language (from the March 1991 *SuperDisk*).

## DEVELOPING WITH SR-INFO

To work through our practical using the *SR-Info* database package, you should make sure you are up-to-date with changes that we made last month on our Magazine Articles project.

We found that the program we created to list articles recorded in the database needed modification to be flexible enough to use all three different key sequences; instead of our original plan (a subprogram called LISTITEM) we have decided to make a major – but easy – modification. Now we have three versions of the subprogram (one for each key sequence) instead of the one.

Last month we were left with one major bug – it sits in the subprograms LISTITEA (authors sequence), LISTITES (subject sequence) and LISTITET (title sequence). There are also a few more limitations to amend.

The bug is quite easy to see in action; select AUTHOR sequence, press [W], then [Enter] when asked for an author, and see what the database does. It displays details of an article, but who is the author? Apparently, his name is 'W'. This applies to all other display screens that follow.

How did you attack this problem? Here's our solution:

```
DO WHILE AUTHOR=FINDER .AND. .NOT. EOF()
    @ 10,18 SAY AUTHOR
    @ 11,10 SAY BLANK(31)
    @ 14,10 SAY BLANK(27)
    @ 11,10 SAY TRIM(FULLNAME#2) + " " + STR(ISSUE,3)
    @ 12,10 SAY "page "+STR(PAGE,3)
    @ 14,10 SAY '"' + TRIM(TITLE) + '"'
    @ 15,10 SAY "Subject.. "+SUBJECT
    ?
    ? "Press any key to continue"
    IF INKEY()=17
    RETURN
    ENDIF
    SKIP
ENDDO
```

This portion of code is the display loop, which is responsible for the screen changing for each article whose author is a match for what we typed; remember, our 'W' is held in FINDER at this point.

Only three lines have been added – the initial three after the DO WHILE command. The first displays the AUTHOR (that missing piece of information), but where? If you look closely, and count to the column after the word 'AUTHOR..' you will see that the contents of the AUTHOR field will be displayed over where we had keyed in the 'W'.

The other two lines improve the display, to get around a little problem

## DESIGNING A DATABASE

A database system can be as simple as a single file – you can write a company name, address and telephone number on an index card, and keep a collection of them in a box, sorted by name into alphabetical order.

If you are keeping more information than that – type of business, main product, and so on – each card will contain a lot of information, yet there will be no easy way to access all the facts. For example, only a long search can tell you the names of the companies specialising in electrical connections.

A useful computer database will usually be made up of several files, rather than one. For instance, a video rental library will want to keep track of which tapes are borrowed by which members – this amounts to matching up two or more sets of things (tapes and members) in different ways.

Many jobs boil down to the same thing – matching passengers to flights (in an airline's system), customers to items purchased (in any invoicing system), students to classes, or companies to products. On the *SuperDisk* we've been developing a database of magazine articles, which relates magazines, authors, and topics.

### STRUCTURE IN A NUTSHELL

In addition to being accessible, a database must be compact and manageable. As we have seen over the last few months, there are several things you can do to organise how information is kept in a database with exactly that goal in mind.

You can design and organise your database in three main steps – 'gather', 'group', and 'refine'.

Let's carefully review these steps, using our Magazine Articles database as an example.

### FIRST STEP – GATHER

Our first step is to gather together all the information that may prove relevant. We aren't gathering the actual data about the magazine articles, instead we're thinking of the sorts of things we want to record.

The magazine database application doesn't just have article titles and magazine names; it has authors, issue numbers, page numbers, and topics. Along with each magazine's name there is its cover price and its editor's name. Some of this information is obviously necessary. Some may seem unnecessary – until you think of what uses could be made of the database.

It's probably a good idea at the outset to include the magazine's address – we may want to write to the editor, and it would be useful to have this information ready to be fed into a word processor. Even if we don't intend to use all this data immediately, it can't do much harm to anticipate future needs.

As we have seen, we should assign short names for each of these sorts of information, decide on their size and their 'type' – text for names, addresses, and similar information, figures for quantities. (Some database languages have other types – dates, logical, and so on). The assigned name, size and type is then called a 'field' within the structure – but we're not finished there.

### SECOND STEP – GROUP

Our next step is to group the different types of information in sensible ways;

## STARTING A LISTING WITHOUT WASTING PAPER

**A little tip, if you want to make sure your '@ SAY' numbering is aligned before you print, but don't want to waste a page by throwing to top-of-form with the EJECT command; look at these three lines, and see what they do**

```
SPOOL RUBBISH.BAK
EJECT
SPOOL
```

**It doesn't matter whether you have SET PRINT ON or not, because EJECT will always go to the printer – or to a spooled file. This bit of code makes sure that you don't lose that blank page. True, it creates a small file, but it is one that can be cleared off your disk at any time.**

with the TRIM function that we used last time. You see, if a long article title is followed by a short one, we could be left with bits of the long one still on the screen – as if it were behind the short one. These two lines blank out the previous displays in certain areas (the magazine name and the article title lines) to prevent that happening.

### A FEW MORE FUNCTIONS

Here is a new function to add to our repertoire; BLANK(31) acts as if it were a constant, 31 characters long, made up entirely of spaces. Of course, you could use just about any number between the brackets – or even a variable. It can take a second parameter in the brackets – a number referring to an ASCII character. The ASCII character for [*] is 42, so BLANK(31,42) will act as if it were a string of 31 stars.

If you didn't have an ASCII chart in front of you, you could have specified BLANK(31,ASC('*')) to get the same effect. Try this out, using *SR-Info's* 'conversational' mode to get an immediate response on screen:

```
NUM=10
?BLANK(NUM,ASC('*'))
```

This will produce a row of ten stars.

Now back to our project. Unless you habitually use Caps Lock, you have probably realised that the text we put into the original ARTICLES database file was all uppercase, and if you search for an author [w] (lower case) you won't find one.

This is easy to correct – again, with a function. Like all functions, this one uses parentheses to surround the information it needs for working; unlike most others, the parentheses are preceded by a single character instead of a meaningful string of characters. Again, [!] ('string') should be treated as if it were a constant string itself – why not print it, using the [?] (immediate print] command?

```
?!('string')
```

The resulting display is the word STRING (in uppercase). If you don't like using the exclamation mark, by the way, you can use the word UPPER as the function name.

You can use this function to good advantage in this subprogram; remember where we TRIM the FINDER

```
$()

Substring function.

Form: $(<string>,<start>,<length>)

Type: CHARACTER

In: <string> - A string expression; the source of the substring.

<start> - A number: a character position in <string>

<length> - A number: the length of the substring.

Out: A substring from the source string <string>, beginning from the
     <start> character position and <length> characters long.

See also SUBSTR()

Previous Screen - <PgUp>, Exit with <Esc>, or Enter command.....
          Enter topic
```

● *SR-Info* has a comprehensive HELP facility, available to the programmer in conversational mode, and while developing programs

each group of fields will form the basic structure of a database file (remember that there may well be more than one file in the database).

The cover price, address and editor's name really have more to do with the magazine than they do with an individual article. It's quite likely that this information will be repeated for many different articles. Therefore, rather than repeat the information over and over again, it would be much better to group the magazine-related fields together into a different file altogether.

When you think of how the database will eventually work, you can start to see the advantage of having two separate files – one with a structure made up of article-related fields, and one consisting of magazine-related fields. This system keeps all the information in a much more compact way.

It also makes changing things a great deal easier; if a magazine moves its office, it makes sense to change the information in just one place, rather than in the record of every single article that has appeared in the magazine.

Of course, there must be provision for some way to link records between the two files; otherwise you couldn't (for example) write to the editor about a particular article – the TITLE field is found in the articles file structure, but the necessary EDITOR field is in the magazine file structure.

We could make a third file – each record holding details of the authors from whom those articles come. Creating the structure for his file can make the whole database more compact; the author's qualifications, for instance, need only exist once (on a single record of an AUTHORS database file) rather than many times (on several records in the ARTICLES database file).

## THIRD STEP – REFINE

Now we must refine the structures of the files in the database. As we have seen, there is a powerful way to avoid needless repetition of data – we simply invent a new file and establish a link to its records; then any number of article records within the database can use the same data, treating it as a record of that new file.

A single record in the ARTICLES file will be related to a single record in the MAGFILE database, for example. The one piece of information that is common to both these two files is the name of the magazine. This particular bit of information must appear as a field in both files – that is in ARTICLES as a 'linking field' and in MAGFILE as a 'key field'. Simply knowing the magazine's name gives you an instant pointer to all sorts of other information about the magazine – from the specific record in MAGFILE.

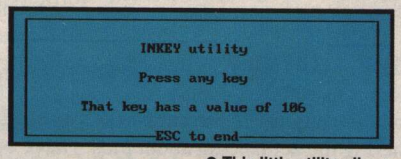
## FINE TUNING

There are still further refinements that can reduce the size of the database. Here are a few suggestions:

Firstly, we can reduce the size of our

## BUILDING AN 'INKEY' UTILITY

**It is quite easy to create a little program that confirms the value of keys, as understood by the INKEY function. Why not code one as an exercise? If you find that you can't work it out, study this for ideas:**




● This little utility allows the database programmer to test values returned by the INKEY function; he can actually use this to build in tests for control keys into database programs.

```
* Here is where we define the window
* set RO to desired row, CO to desired column
RO = 5
CO = 33
********************
ON escape
   WINDOW
   CLS
   RETURN
   ENDON
   CLS
   WINDOW RO,CO,RO+6,CO+40
   @ RO+1,CO say cen("INKEY utility",40)
   @ RO+3,CO say cen("Press any key",40)
   esctoend="ESC to end"
   @ RO+7,CO+20-len(esctoend)/2 say esctoend
   **********************
   DO WHILE .t.
   @ RO+5,CO say cen("That key has a value of "+str(inkey(),3),40)
   RING
   ENDDO
```

field, once it has been keyed, to get rid of trailing spaces? You can combine this function with TRIM, to capitalise what was typed at the same time.

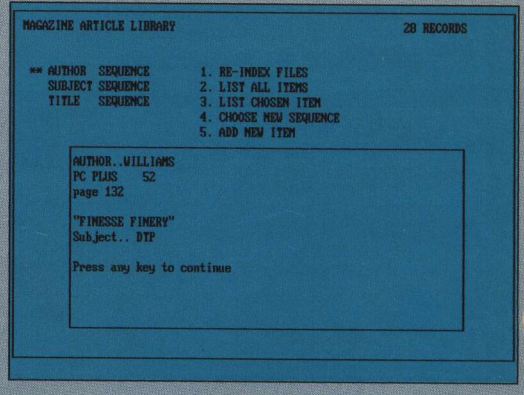
```
FINDER=!(TRIM(FINDER))
```

This shows, by the way, that you can combine several functions one within another, but be careful to get the parentheses in the right place.

Another *SR-Info* function that will make this routine a lot more useful is INKEY(). It works just like a WAIT instruction, stopping the program until a key is pressed – but it then acts as if its value were a number. That number can be used to tell what key was pressed. Ordinary keys produce the ASCII number of the character – the [*] symbol will produce 42, for example. The other keys to note are the F1 to F10 keys (numbers 315 to 324), Home (327), End (335), Insert (338), Delete (339), up (328), down (336), left (331), right (333), enter (13) and [Ctrl Q] (17).

When you look to the bottom of this display loop, you will see that the old WAIT command has been substituted by a little IF/ENDIF structure that uses the INKEY function. What does this do? Incorporate the change, then DO ARTILIST to try it out.

Previously we suggested how the LIST ALL ITEMS operation could work – we are getting very close to coding it all for you! There are plenty of functions – among those we haven't yet used, are LOWER (opposite of UPPER), SUBSTR, LEFT, RIGHT, and PIC. These are explained within *SR-Info's* HELP system – the best way to learn is to try experimenting with them.




● The very slightly revised operation of our Magazine Articles application, interrogating the database in AUTHOR sequence.

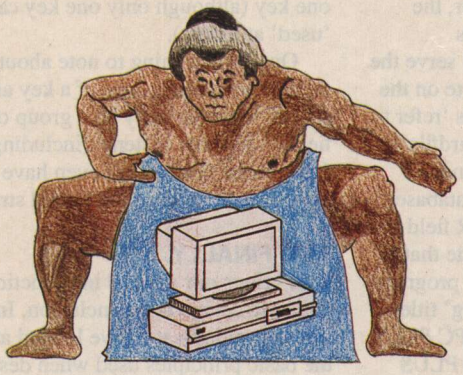
## NON-SCREEN OUTPUT

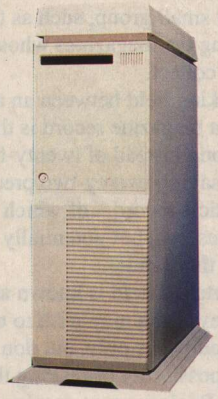
One of the most useful things about a database is its ability to supply information for lists, so let's take a look at printing with *SR-Info*. So far we have used '@ SAY' to put things to a specific place on-screen and we do the same for printing, with just a few changes.

The first thing to note is that *SR-Info* starts as if SET PRINT OFF had been

# THE LITTLE BIG MACHINE

SUMU





| 286 - 12MHz | 386SX - 16MHz | 386 - 25MHz | 486 - 33MHz |
|---|---|---|---|
| £399.00 | £599.00 | £799.00 | £1999.00 |

## All basic systems are supplied with:

- 1 MB RAM
- Single floppy disc drives 3½" or 5¼"
- Desktop or mini tower case and PSU
- 14" monochrome monitor
- Keyboard
- Serial, Parallel and Controller cards
- 1year on-site maintenance
- Unlimited support and advice

| BASIC SYSTEMS | | PRICE |
|---|---|---|
| SPC1-286 | -12 MHz. | £399 |
| SPC2-286 | -16MHz. | £499 |
| SPC3-386sx | -16MHz. | £599 |
| SPC4-386 | -25MHz. | £799 |
| SPC5-386 | -33MHz. | £999 |
| SPC6-486 | -25MHz. | £1499 |
| SPC7-486 | -33MHz. | £1999 |

| OPTIONAL EXTRAS | PRICE |
|---|---|
| Floppy Disk Drive | £50 |
| Mono VGA | £80 |
| Colour Super VGA | £205 |
| 1MB Ram | £50 |
| 40 MB Hard Disk Drive | £150 |
| 80 MB Hard Disk Drive | £200 |
| Windows, Dos & Mouse | £100 |

Please call for other items and peripherals
All prices are exclusive of V.A.T


**SUMU COMPUTERS**

**17 SPLOTT ROAD,**
**CARDIFF CF2 2BU**
**TEL: 0222 470460 / 0222 482741**
**FAX: 0222 482724**

Member of PEPLOW (UK) Ltd group


Please supply me with information/system for:
.....
.....
.....
Name:.....
Position:.....
Company:.....
Address:.....
.....
.....
Telephone:.....
Fax:.....
I enclose cheque/card number:.....
for £:.....

BARCLAYCARD VISA

linking fields and keys – perhaps by using a nickname, or a number. Surprisingly, this can even be a help with a fairly small group, such as the choice among ten magazines whose articles you collect.

If the linking field between an article record, and a magazine record is three characters long instead of twenty-five, you will be saving twenty-two precious bytes per article record – of which there will not be just ten, but potentially hundreds or thousands.

This shortened field is known as an 'artificial' key – you may wish to use it for internal purposes only, but don't dismiss the possibility of making this key official. Bank account numbers, for example, are actually artificial keys. (It's important to remember that by 'artificial' we mean made up – we do not necessarily mean fictitious).

We can also invent nicknames for fields that have a limited, repeated range of values. For example, the FREQUENCY field within the MAGFILE database could easily be abbreviated to W for weekly, M for monthly, Q for quarterly, and so on.

This nickname can then be used as a linking field into a database which only has its key, plus the expanded meaning (for example – the key 'Q', with the expanded meaning of 'Quarterly'). Don't be afraid to create such a tiny database – it can save you space, and still give you the same full information without extra effort.

## MORE KEY INFORMATION

When used to link files together, the linking field (in one file) and its matching key (in the other file) serve the same purpose as a pencilled note on the bottom of an index card such as 'refer to card 432 in the PRODUCTS cardfile'.

A key can also be used to find particular records within one database file. If you have a PUBLISHER field on your magazine file and designate that as a key, you could have a simple program search for all 'Future Publishing' titles, and find among them not only PC PLUS, but also AMIGA Format, 8000 PLUS and others.

The third use of keys makes it easy to produce reports in any particular order. If you wish to print a list of articles grouped together in topic sequence, the report program needs only to read the articles database using the TOPIC index – a tiny file of pointers to the articles, sorted into good order by using TOPIC as a key-field.

We saw that a key field – for file-linking purposes – should have a unique value, but with these two new uses, there is no such restriction. When finding records (as in 'on line enquiry') or grouping records (for reports) we can cope quite easily with the same value appearing in the key field of two or more records. Be careful, sometimes a key will be used for more than one of these three ways. And do not forget – a database file can have more than just the one key (although only one key can be 'used' at a time).

One further thing to note about keys; though we have spoken of a key as being a field, it can actually be a group of fields – in some systems (including *SR-Info),* those fields don't even have to be next to each other in the record structure.

## AND FINALLY...

This brings our general introduction to database design to a conclusion. In the last four months we have looked at all the basic principles used when designing a database structure, but there are many specialised subjects that deserve discussion in depth at a later time. For example, the use of a database file – how rapidly its contents change – has a bearing on efficient design. From time to time in future issues we will look at these subjects.

Armed with these introductory studies, you will now have the means to design working database systems. We are interested in your ideas, suggestions, and results – please send us your work. The best *SR-Info* examples – whether modifications to the skeletal MAGAZINE ARTICLES database, or entirely new applications – will appear on future *SuperDisks*. ●

coded, and so all '@ SAY' commands send output to the screen. You can change this simply by sending the command, SET PRINT ON.

You should also note how to force a new page on the printer – the command EJECT will do the job, but don't try it if you are going to the screen.

You can even 'print' when your printer is not available simply by using the SPOOL command. SPOOL MYFILE, for example, will open a disk file called MYFILE.TXT – and if we SET PRINT ON, the output will go neither to the screen, nor the printer, but to MYFILE.TXT (you stop this with SPOOL – no parameter). You can even print MYFILE.TXT subsequently – when the printer is available, by executing the command SPOOL MYFILE.TXT TO 1 (the number is the printer port being used).

## UPDATING INFORMATION

Modifying database information safely is essential. There are few file commands within *SR-Info;* this is intentional – so that individual commands are less likely to corrupt the relationships set up so carefully. We have seen GO TOP, GO BOTTOM and SKIP – all commands that position you to a new place within the database. If you know the number (that is, position within the database) of a particular record, you can specify this with the GO command, but it is rare that this is useful. Perhaps it can be used to resume at a certain place, after being interrupted for some other operation.

There is no 'rewrite' command – but records can be modified with a special kind of 'move' command. As we have seen, memory variables can be modified by resetting their value with an equal-sign operation; examples are:

```
COUNTER = COUNTER + 1
NEWMESSAGE = "NO PROBLEMS"
```

You can do the same with database fields on the left-hand side of these expressions, but you only change the record temporarily, while it is in the computer's memory – it doesn't get changed on the disk. However, let's suppose COUNTER and NEWMESSAGE are actually database fields. We can make the change permanent by coding:

```
REPLACE COUNTER WITH COUNTER + 1
REPLACE NEWMESSAGE WITH "NO PROBLEMS"
```

We use REPLACE not only to change existing records in the database, but also for adding records as well. You see, the most secure way of adding a new record to the database is to use a simple command, APPEND BLANK. This adds a record to the active database (the one pointed to by the last SELECT), points to it, and you are now ready to REPLACE its fields.

## ALTERNATIVES

There is one other way to make permanent changes to the active database, but many programmers shun this method. The '@ GET' operation also makes permanent changes, like the REPLACE command. It is more common to pick up keyed values with '@ GET', but place them into memory (and therefore temporary) variables first, and REPLACE the database fields with the memory variables at a later stage. Remember that all the '@ GET' operations are put into action one-by-one when the READ command is executed – but things may 'go wrong' during the keying, and you may wish to stop any updating that was started. Yet if you did a '@ GET' into a database field earlier, it's too late.